Chapter 3
Efficient Range Query Processing on Complicated Uncertain Data

Andrew Knight, Qi Yu, and Manjeet Rege

Abstract Uncertain data has emerged as a key data type in many applications. New and efficient query processing techniques need to be developed due to the inherent complexity of this new type of data. In this paper, we investigate a special type of query, range queries, on uncertain data. We propose a threshold interval indexing structure that aims to balance different time consuming factors to achieve an optimal overall query performance. We also present a more efficient version of our structure which loads its primary tree into memory for faster processing. Experimental results are presented to justify the efficiency of the proposed query processing technique.

Introduction

The term *uncertain data* defines data collected with an inherent and distinctly quantifiable level of uncertainty [2]. Whereas values for *certain data* are given as exact constants, values for uncertain data are instead given by probability measures, most notably by probability distribution functions (PDFs) [1, 10, 11, 25, 27]. Uncertain data has been increasingly generated from a great variety of applications such as:

- Scientific measurements include margins of error due to limited instruments [2].
- Sensor networks are imprecise due to hardware limits [2].
- GPS is only accurate to within a few meters [11].
- Uncertainty in mobile object tracking can magnify errors in predictive queries [2, 34].
- Forecasting weather or economic data is based heavily upon statistics and probabilities [3, 25, 30].
- Aggregated demographic data only represent summaries, not actual data [4].

A. Knight (✉) · Q. Yu · M. Rege
Rochester Institute of Technology, One Lomb Memorial Drive, Rochester, NY 14623-5603, USA
e-mail: andy.knig@gmail.com; qyu@it.rit.edu; mr@cs.rit.edu

- Privacy-preserving data mining often introduces jitter to protect individuals [2].
- Lost information creates incompleteness in data [25].

In contrast to certain data whose values are exact constants, uncertain data take values that are described by probability measures, most notably probability distribution functions (PDFs). Due to the inherent complexity of uncertain data, new techniques need to be developed in order to efficiently process queries against this new type of data. A set of novel indexing structures have been developed to accelerate query processing. Representative ones include threshold index [10, 26], the U-tree [30], and 2D mapping techniques [1, 10]. Most of these approaches assume that disk I/Os are the dominating factor that determines the overall query performance. Thus, the indexing structures are usually designed to optimize the number of disk I/Os. However, uncertain data is inherently more complicated than certain data. Computing a range query on uncertain data usually involves complicated computations, which incur high CPU cost. This makes disk I/Os no longer the solely dominating factor that determines the overall query performance. Therefore, new indexing strategies need to be developed to optimize the overall performance of range queries on uncertain data.

Uncertain continuous data is usually modeled by PDFs. Some PDFs, like the uniform PDF, may be very simple to calculate. However, many widely used PDFs involve complicated computations, such as multimodal probability models for cluster analysis [33] or computer simulations of cell signaling dynamics for biology research [21]. Computing a complicated PDF may involve high computational cost. Numerical approaches, such as Monte Carlo integration, have been exploited to improve the performance [30]. Riemann sum, however, provides a better strategy for one-dimensional cases. Even though Riemann sums can be faster than Monte Carlo integrations, they still incur high computational cost. Especially when a high computation accuracy is required, probability calculations may take even longer than disk I/Os. Therefore, the number of probability calculations must also be considered if the distribution of the uncertain data is complicated. In this case, the indexing strategy needs to balance between disk I/Os and the CPU cost to achieve an optimal overall query performance.

In this paper, we present a novel indexing strategy focusing on one-dimensional uncertain continuous data, called *threshold interval indexing*. It addresses the limitations of existing indexing structures on uncertain data, particularly for handling complicated PDFs, by treating uncertain objects as intervals and thereby leveraging interval tree techniques. The proposed indexing structure is also inspired by the optimized interval techniques from [7] to build a dynamic primary tree and store objects in nodes at different levels depending on the objects' sizes. The notion of using an interval tree to index uncertain data was suggested by Cheng et al. in [10] but disregarded in favor of an R-tree with extra probability limits called x-bounds. We assert that x-bounds can just as easily be applied to interval trees to index uncertain data with special benefits. We also propose a *memory-loaded threshold interval index*, which loads the primary tree into memory for faster processing.

The rest of the paper is organized as follows. Section “Background and Motivation” gives the problem statement and provides an overview of previous research. Section “Processing Range Queries on Uncertain Data” presents the threshold interval index. Section “Memory-Loaded Threshold Interval Indexing” introduces the memory-loaded version of the threshold interval index. Section “Experimental Results” gives experimental results of our two indexes versus the probability threshold index. Section “Conclusion” concludes the paper by offering direction for future research.

Background and Motivation

Existing indexes are inadequate for handling complicated uncertain continuous data. This section will give the problem statement, describe existing indexing strategies along with shortcomings, and provide an overview of related work.

Problem Statement

Given a database table T, a *query interval* $[a, b]$ for an uncertain attribute e of an object u_i, and a *threshold probability* τ, a *range query* returns all uncertain objects u_i from T for which $Pr(u_i.e \in [a, b]) \geq \tau$.

Theoretically, an uncertain object could have more than one uncertain attribute. However, for this paper, we focus on indexing objects based on only one uncertain attribute.

With no index, a query must calculate a probability for each object to determine if the object falls within the query interval. Naturally, an efficient index prunes many uncertain objects from a search to avoid unnecessary probability calculations, which, given the complexity of the PDFs, could save a lot of time.

External Interval Tree Index

Interval trees [12] are not specifically designed for handling uncertain data, but one-dimensional uncertain objects may be treated as intervals by using their PDF endpoints. Arge et al. [6] propose two optimal external interval tree indexes. Both indexes use a primary tree for layout and secondary structures to store the objects at each node. The first index’s primary tree is a balanced tree over a set of fixed endpoints with a branching factor of $\sqrt{B}$ as the base tree, where B is the block size. The second index replaces the static interval tree with a *weight-balanced B-tree* [6] storing interval endpoints to achieve dynamic interval management. Note that its primary tree does *not* store the intervals, it only stores endpoints to control tree spread. In both indexes, each internal node v represents an interval I_v containing all of its child nodes' endpoints. Each interval I_v is divided into subintervals called *slabs* by the endpoint boundaries on v's immediate child nodes. When using this tree to index a set of interval objects I, an interval $i \in I$ is stored at the lowest node v in the tree such that i is not split across slab boundaries. Each node v stores these intervals in secondary structures for each slab boundary: B-trees normally or in an underflow structure if the number of segments is less than $B/2$ [6, 17]. These lists hold all intervals that cross the boundary on the left side, on the right side, and as a multislab. *Stabbing queries* are used to return results.

Since the endpoints in the first index are fixed, it can become unbalanced and therefore inefficient due to spread and skew in the input interval set. The second index, although much more complicated, adapts well to skew and to new inputs. However, the downfall of both interval indexes, as mentioned in [10], is that if many uncertainty intervals overlap with the query interval's endpoints, then few objects are pruned from the search, and a lot of time is wasted in calculating probabilities. Furthermore, although this external index is theoretically optimal, it is not always practical [23].

Probability Threshold Index

The *probability threshold index* (PTI) [10] allows range queries to prune more branches from searching than interval indexes allow. The PTI uses a one-dimensional R-tree as a base tree. Only leaves store uncertain objects. Each internal node has a *minimum bounding rectangle* (MBR) that encloses the narrowest boundaries $[L, R]$ for all child PDFs. Tighter bounds, called *x-bounds*, are also calculated for each node. X-bounds are the pair of boundaries (L_x, R_x) such that the probability an object attribute's value exists in $[L, L_x]$ or $[R_x, R]$ is equal to x [10]. Thus, when performing a range query for objects in $[a, b]$ with probability threshold τ, if, for a certain node, $\tau \geq x$ and $[a, b]$ does not overlap L_x or R_x at its right or left ends, then the node and its children may be pruned from the search.

The PTI has many advantages. It is an elegant solution, and it is fairly easy to implement. The tree is dynamic as well. All boundaries are calculated when objects are added. Multiple x-bounds can be stored in each node, so queries can choose the most appropriate bounds for its threshold. Required storage space for internal nodes is relatively small. Note that the U-tree is very much like a multi-dimensional PTI with additional pruning techniques [30].

The PTI is not without weaknesses, however. The primary weakness pointed out by Cheng et al. is that differences in interval sizes will skew the balance of the tree [10]. Methods involving *variance-based clustering* are provided in [10] to solve this problem; however, they only work for PDFs that are *variance monotonic*. Furthermore, Cheng et al. do not provide an optimal rectangle layout strategy for the PTI's base tree, the R-tree. The best strategy for any R-tree is to make MBRs as disjoint as possible. When MBRs overlap too much, extra disk I/Os and probability

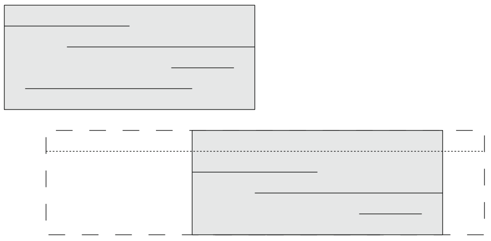
Fig. 3.1 MBRs can easily become skewed. The *dotted rectangle* shows how the bottom MBR must expand to accommodate the uncertain object denoted by the *dotted line*. These two MBRs now severely overlap

calculations must be performed because fewer nodes can be pruned. Adding new objects, especially objects of vastly different interval lengths, exacerbate overlap, as shown in Fig. 3.1. Simply put, sloppy R-trees are inefficient, but optimal R-trees are very difficult to maintain. Strategies such as segment indexes and the SR-tree [18] address different interval lengths, and interval indexes handle skew very well.

When rectangles overlap, not all objects which fall completely within the query interval can be immediately accepted. Since MBRs might overlap, every node must be checked. There is no exclusivity between node intervals. Nodes may not be stored in any order if their intervals are stretched. Objects might appear in the overlapping portions of nodes, too. These compounding factors force probability calculations on all objects in each unpruned node. This wastes lots of time, especially when the query interval is much larger in size than most uncertainty intervals.

2D Mapping Indexes

Cheng et al. first suggested 2D mapping techniques as an alternative to the PTI for uniform PDFs [10]. Agarwal et al. then expanded 2D mapping techniques to histogram PDFs [1]. Histogram PDFs can easily be transformed into linear piecewise cumulative distribution functions (CDFs). A CDF F can then be transformed into a linear piecewise threshold function g, for which $g(x)$ gives the minimum value y such that $F(y) - F(x) \geq \tau$ for a preset probability threshold τ. Threshold functions are calculated for each uncertain object and turned into a set of line segments. A range query for an interval $[a, b]$ graphs the point (a, b) and returns all objects whose line segment threshold functions are below it.

The structures of the indexes presented in [1] manipulate the line segments. The half-plane range reporting technique partitions the line segments into sets of layers. Queries visit each layer in order until a layer surpasses the query interval. Fractional cascading improves visit time. The segment tree, interval tree, and hybrid tree use the same notions presented in [6] about interval management and slabs to form optimized index structures.

2D mapping indexes are efficient for uniform and histogram PDFs, but they are inapplicable for more general PDFs. Furthermore, each index is rigidly based upon one threshold value; separate indexes must be constructed for additional thresholds.

This is starkly different from the PTI, which can manage several threshold values in one structure. However, the application of interval tree techniques presented in [1] is a novel enhancement over techniques presented in [6].

Related Work

Ranking Queries

A few different types of ranking queries have been proposed for uncertain data, which use the probabilistic database model. U-Top$k$ queries return the list of k tuples with the highest probability of being ranked as the top k [25, 29, 32]. To efficiently perform a U-Top$k$ query, a search of possible states is performed, in which the search is extended only for the tuples of highest probability. A U-kRanks query return the tuple with the highest probability of being ranked at each position. This implies that the same tuple could appear in more than one position [25, 29, 32]. When evaluating a U-kRanks query, only the most probable state for each rank so far needs to be stored. Independent tuples also exhibit the optimal substructure property, meaning a dynamic programming solution is possible. The probabilistic threshold top-k (PT-k) query returns all tuples which have a probability greater than some threshold probability for being ranked in the top k positions [15, 15, 25]. This captures tuples which might be missed by a U-Top$k$ or U-kRanks query. To evaluate, the dominant set property is leveraged, which states that whether or not a tuple is in the result set of the query depends on how many other tuples are ranked higher. Generation rule compressions and pruning improve query time. Sampling methods and Poisson approximation methods can improve efficiency in exchange for accuracy [15].

Joins

A join between tables with uncertain data returns a cross product in which each paired tuple is associated with a probability $p \geq 0$. A *probabilistic join query* (PJQ) between two tables returns all pairs of tuples with a non-zero probability of meeting the join condition. Likewise, a *probabilistic threshold join query* (PTJQ) only returns tuples whose probability is greater than some threshold probability τ. A *confidence-based top-k join query* (PTop$k$JQ) returns the k tuples with the highest probability resulting from the join. It is possible to apply various existing join methods in addition to pruning techniques to optimize query time [11, 20]. Three primary techniques for joining uncertain continuous data are presented in [11]. Joins on discrete data operate differently. For the *similarity join query*, [19, 20], each uncertain object is turned into a vector of its possible attributes. These vectors are then clustered into groups using the k-means clustering algorithm, and each group is approximated by calculating its minimum bounding hyper-rectangle.

Skyline Searches

Skyline analysis of a data set searches for the "best" objects by weighing tradeoffs between attributes. The most desirable objects constitute the *skyline*. Performing skyline searches on uncertain data can also be useful [24]. Pei et al. give the example of data representing NBA players' statistics. Reverse skyline searches can also be applied to uncertain data [22]. A reverse skyline obtains a dynamic skyline based on query parameters [13]. In a sense, it can find lowers layers of skylines instead of just the top layer. Reverse skylines are useful, particularly with uncertain data, when verifying faulty equipment or abnormal data [22]. A monochromatic probabilistic reverse skyline queries find reverse skylines over one data set, and bichromatic queries find reverse skylines between two data sets. Pruning can be performed spatially and probabilistically for both approaches. Offline pre-computation of pruning spaces can optimize queries further.

Indexing Categorical Data

Categorical (discrete) data has a finite data domain $D = \{d_1..d_n\}$. Each uncertain attribute within an object is called an *uncertain discrete attribute* (UDA) u. A UDA's value might be any value in the data domain. Therefore, it is represented by a vector of probabilities $u.P = \langle p_1..p_n \rangle$ such that $Pr(u = d_i) = u.p_i$. The probability that two UDAs are equal is given by the dot product of their probability vectors. Effective indexing strategies make equality queries and ranking queries more efficient [26, 27]. Two strategies are proposed by Singh et al. [27]. The first is the *probabilistic inverted index*, and the second strategy is the *probabilistic distribution R-tree* (PDR-tree). Testing shows that there is no universal winner between the two strategies [27].

High Dimensionality

The strategies discussed for continuous and categorical uncertain data focus on indexing one dimension. Sometimes, however, it makes more sense to think of uncertainty in more than one dimension. For example, spacial data, like for GPS or for location-based services [31], has a region of uncertainty, not just values along one line. The U-tree [30] is a natural extension of the probability threshold index [10]. Instead of using an R-tree as the base, the U-tree uses an R*-tree, which is a multidimensional R-tree that optimizes its structure by minimizing area, margin, overlap, and distance of minimum bounding rectangles [9]. Instead of using just MBRs, the U-tree uses *probabilistically constrained regions* (PCRs) to tighten regions, much like x-bounds for the probability threshold index, based on probability threshold values, and can be used both to prune and to validate results without further PDF calculation. A major problem with multidimensional data is the *sparsity problem*: distances between pairs of points tend to be too similar to garner quality information using standard distance functions [4]. Aggarwal et al. propose an expected distance function which uses a contrast ratio between means and standard deviations for a fraction of the uncertain objects [4].

Moving Objects

Indexing moving objects has been well researched [16, 34]. An example of indexing moving objects would be for a city bus route schedule: busses drive along the streets, and they can be tracked to give real-time feedback on their arrival times. For certain moving data, a robust indexing structure is the B^x-tree [16]. Adding uncertainty to the data model for moving objects allows for more accurate models [34].

Processing Range Queries on Uncertain Data

We now present the proposed range query process technique for uncertain data. The cornerstone is the *threshold interval index* (TII). The TII is in essence a combination of a dynamic external interval tree and the x-bounds structure used by PTI. This structure presents two key advantages. The first advantage is that the structure intrinsically and dynamically maintains balance all the time. The second advantage is that the interval-based structure makes all uncertain objects which fall entirely within the query interval easy to find and, therefore, possible to add to the results set without further calculation. The PTI does not allow this because its MBRs might overlap. Furthermore, adding x-bound avoids the interval index's problem for when many uncertainty intervals overlap the query interval.

TII Structure

The TII has a primary tree to manage interval endpoints. It also has secondary structures at internal nodes of the primary tree to store objects. When an object is added to the index, the endpoints of its uncertainty interval are added to the primary tree. Then, the object itself is added to the secondary structures of the appropriate tree node. Each object is also assigned a unique id if it does not already have one. X-bounds are stored for each internal node.

Primary Tree

The primary tree is a *weight-balanced B-tree* with branching parameter $r > 4$ and leaf parameter $k > 0$. The *weight* of a node is the number of items (in this case, endpoints) below it. All leaves are on level 0. All endpoints are stored at the leaves,

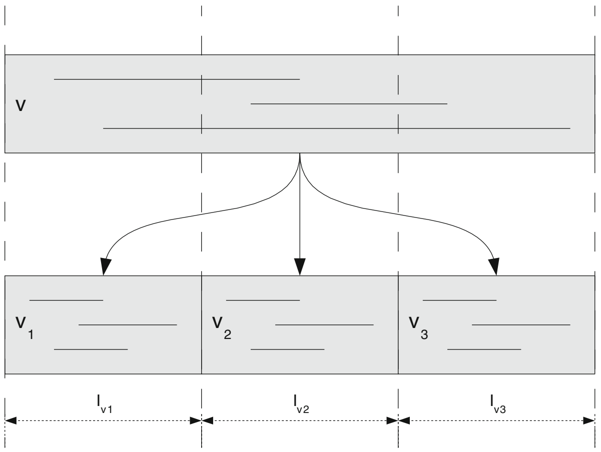

Fig. 3.2 A node v with three child nodes. The *dotted lines* denote slab boundaries. Note how objects are only stored within intervals which can completely contain them

and internal nodes hold copied values of endpoints. The weight-balanced B-tree provides an effective way to dynamically manage intervals and spread. Arge et al. describe this tree in detail, including time bounds, in [7].

Secondary Structures

Each internal node v represents an interval I_v, which spans all interval endpoints represented by children of v. Thus, the c children of v (for $\frac{1}{4}r \le c \le 4r$) naturally partition I_v into subintervals called *slabs* [7]. Each slab is denoted by I_{v_i} (for $1 \le i \le c$), and a contiguous region of slabs, such as $I_{v_2}I_{v_3}I_{v_4}$, is called a *multislab* [7]. All slab boundaries within I_v are stored in v. Note that I_{v_i} is the interval for the child node v_i.

An uncertain object is stored at v if its uncertainty interval falls entirely within I_v but overlaps one or more boundaries of any child node's I_{v_i}. (A leaf stores uncertain objects whose PDF endpoints are contained completely within the leaf's interval endpoints.) Each object is stored at exactly one node in the tree, as shown in Fig. 3.2. Let U_v denote the set of uncertain objects stored in v. In the external dynamic interval index, these objects are stored in secondary structures called *slab lists* [7], partitioned by the slab boundaries. However, only two secondary structures are needed per node for the TII because range queries (described later in this section) work slightly differently than stabbing queries. The *left endpoint list* stores all uncertain objects in increasing order of their uncertainty intervals' left endpoints. The *right endpoint list* stores all uncertain objects in increasing order of their uncertainty intervals' right endpoints. This is drastically simpler than the optimal external interval tree, which requires a secondary structure for each multislab [7].

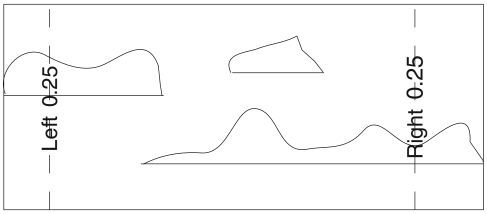

Fig. 3.3 A node with x-bounds and objects. X-bounds are calculated for each node based on uncertain objects' PDFs. The *left* and *right* 0.25-bounds are tighter than the MBR

If the uncertain objects hold extra data or large PDFs, it might be advantageous to store only uncertainty interval boundary points and object references in the two lists. The actual objects can be stored in a third structure to avoid duplication.

Applying X-Bounds

X-bounds were introduced as part of the probability threshold index [10] and can easily be applied to the TII.

Definition 1. An *x-bound* is a pair of values (L_x, R_x) for a continuous PDF $f(s)$ with uncertainty domain $[L, R]$ such that

$$x = \int_{L}^{L_x} f(s)ds = \int_{R_x}^{R} f(s)ds \tag{3.1}$$

L_x is the *left x-bound*, and R_x is the *right x-bound*. Since the domain of $f(s)$ is $[L, R]$, L_x and R_x are unique. Note that x is a probability value, meaning $0 \leq x \leq 1$. For example, if $x = 0.25$, then there is a 25 % chance that the object's value appears in the interval $[L, L_{0.25}]$. Furthermore, there would be a 25 % chance it appears in $[R_{0.25}, R]$ and a 50 % chance it appears in $[L_{0.25}, R_{0.25}]$. Note also that if $x > 0.5$, then $R_x < L_x$.

The notion of x-bounds can be applied to tree nodes as well as to PDFs, as seen in Fig. 3.3. The left x-bound for a node is the minimum left x-bound of all child nodes and objects, and the right x-bound is the maximum right x-bound of all child nodes and objects. Specifically, for a node v, left and right x-bounds are calculated for I_v. A child node's x-bounds must be considered when calculating v's x-bounds: a child node might have tighter x-bounds than any of the uncertain objects stored at v. The interval I_v accounts for all uncertain objects stored at v and in any child nodes of v, and so should the x-bounds. The x-bounds for v's slabs are given by the x-bounds on v's child nodes. All of v's x-bounds are stored in v's parent. In this way, the interval I_v is analogous to a minimum bounding rectangle in an R-tree, and intervals are tightened by x-bounds in the same way as MBRs are tightened in the PTI [10]. X-bounds for more than one probability x can be stored as well.

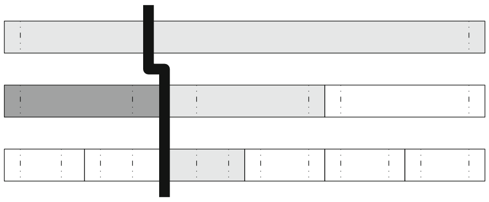
Fig. 3.4 A left stab is denoted by the *thick black line*. The *light gray* nodes are visited by the stab. The *white* nodes are not visited. The *dark gray* node is pruned based on x-bounds

Range Query Evaluation

Evaluating range queries for objects in $[a, b]$ with a threshold τ on the TII is like evaluating stabbing queries on a regular interval tree. Two *stabs* are executed for each endpoint of the query interval: a left stab and a right stab. The nature of the query forces these stabs to be performed slightly differently from how they are described in [7]. Once the stabs are made, a series of *grabs* can be performed for all objects in between. This is called the *stab 'n grab search.*

The Left Stab

The *left stab* is the most complicated part of the stab 'n grab search. The search starts at the root node and continues down one path through child nodes until it hits the leaf containing the closest x-bound to a within its boundaries. This leaf is called the *left boundary leaf.* X-bounds are used to prune this search. If a node's right x-bound is less than a or if a node's left x-bound is greater than b, then the node can be pruned, because the probability that any of its objects falls within the query interval must be less than the query's threshold. Objects are checked at nodes along the stab to see if they belong to the result set (Fig. 3.4).

Before moving to the next child node, the uncertain objects stored in secondary structures at the current node must be investigated, because their uncertainty intervals may overlap the query interval. If they overlap the query interval, then they might be valid query results. Between the secondary structures, only the right endpoint list is needed. A quick binary search can be performed to find which objects fall within the query interval. Any object whose right endpoint is less than a can be disregarded. Any object whose both endpoints are within the query interval is added to the result set automatically. Otherwise, a probability calculation must be performed using the object's PDF to determine if it meets the threshold probability. The same strategy applies for the left boundary leaf. All valid objects are added to the result set.

The Right Stab

The *right stab* is analogous to the left stab, except it searches with b instead of a. The leaf found at the bottom of the stab is called the *right boundary leaf.* X-bound

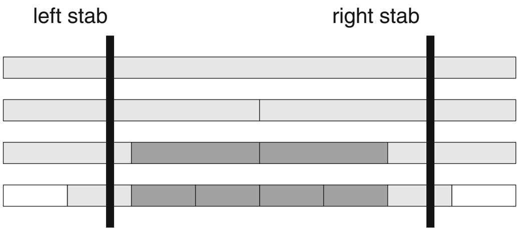

Fig. 3.5 A stab 'n grab query. The *light gray* nodes are visited during the stabs, and the *dark gray* nodes are visited during the grabs. Note how grabbed nodes fall completely within the query interval

pruning is performed for the rightmost child nodes, not the leftmost. The process for searching the secondary structures is the same as in the left stab, except "left" and "right" are switched wherever mentioned. Furthermore, nodes visited during the left stab can be skipped during the right stab, because the process for investigating uncertain objects accounts for both endpoints of the uncertainty interval. This is why references to visited nodes are stored during the left stab.

The Grabs

The two stabs find the two boundary leaves and some uncertain objects in the result set. The remaining objects to investigate reside in the nodes between the two boundary leaves. Thankfully, all objects in between can be added to the result set without any probability calculations. Remember, intervals for nodes on the same level do not overlap, so all objects stored at nodes between the boundary leaves must fall entirely within the query. The most effective way to grab all of these uncertain objects is to perform a post-order tree traversal starting at the left boundary leaf and ending at the right boundary leaf, skipping each node that has already been visited. No extra searching needs to be done on the secondary structures. Figure 3.5 illustrates a full stab 'n grab query.

Time Bounds

A range query can be answered within the following time bounds using the stab 'n grab search:

Theorem 1. *Let I be a TII storing N uncertain objects, whose primary tree has branching parameter r and leaf parameter k. Assume any calculation on an uncertain object's PDF takes $O(d)$ time. A range query Q with query interval $[a, b]$ and threshold τ can return all T uncertain objects stored in I which fall within the query interval with probability $p \geq \tau$ in $O(kd\ log_r(N/k) + T/k)$ time.*

Proof. The height of the primary tree is $O(\log_r(N/k))$ [7]. If the number of child nodes of any internal node is $O(a)$, then the total number of nodes in the tree is $O(\Sigma_{i=0}^{\log_r(N/k)} r^i) = O(r^{\log_r(N/k)}) = O(N/k)$. Since the N uncertain objects are distributed relatively uniformly over the tree, each node stores $O(N/(N/k)) = O(k)$ objects. A stab, either right or left, visits $O(\log_r(N/k))$ nodes from root to boundary leaf and must visit all objects stored at a node in the worst case, calculating probabilities for each. Hence, the stabs are performed in $O(kd \log_r(N/k))$ time. The grabs are performed in $O(T/k)$ time, because extra checking at each node in between the leaf boundaries is unnecessary. All nodes visited by the grabs are guaranteed to be valid results, so T is used instead of N for the time bound. Therefore, two stabs and all grabs can be performed in a combined time of $O(kd \log_r(N/k) + T/k)$. □

Externalization

The TII can easily be externalized by setting k and r for the primary tree appropriately, albeit differently than how described in [7]. Let B be the block size; specifically, the number of data units which can be stored in a block. For the primary tree, an uncertain object is represented only by its uncertainty interval endpoints, each of which is one unit of data. Each child node needs two units for endpoints, one unit for its block pointer, and two units for each set of x-bounds, meaning each node requires $3 + 2n$ units, where n is the number of x-bounds stored for the tree. The number of children per node still ranges from $\frac{1}{4}r$ to $4r$. Thus, $k = \frac{1}{2}B$ and $r = \frac{1}{4(3+2n)}B$.

Theorem 2. *The external TII can be stored using $O(N/B)$ blocks. Range queries can be answered using $O(log_B N + T/B)$ disk I/Os and $O(B\ log_B N)$ probability calculations, and updates can be performed using $O(log_B N)$ disk I/Os and $O(1)$ probability calculations.*

Proof. Substitute $r = \frac{1}{4(3+2n)}B$ and $k = \frac{1}{2}B$. The bounds from [7] still remain for disk I/Os. □

Memory-Loaded Threshold Interval Indexing

Although the TII aptly balances the data, the primary tree uses a lot of storage space. For every object, the primary tree must also store two endpoints. This can severely inflate the number of blocks when many objects are indexed externally. In this section, we introduce the *memory-loaded threshold interval index* (MTII) as an alternative external TII to reduce the number of disk I/Os during range queries. Since the primary tree is significantly smaller, all nodes can be preloaded into memory before the query runs to improve runtime.

In the TII, every uncertain object's endpoints are stored in the leaves. However, in the MTII, only those endpoints which form the slab boundaries, e.g., the minimum

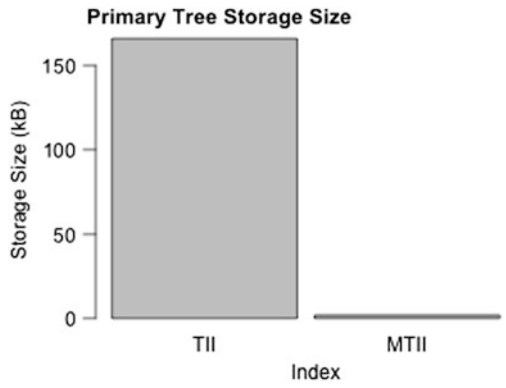

Fig. 3.6 Size comparison for 10,000 objects and 1 X-bound

and maximum values for each leaf's interval, must be stored. This means that a leaf stores only two endpoints instead of k endpoints. An internal node stores its own slab boundaries and pointers to child nodes. It does not need to store the slab boundaries or x-bounds for its child nodes.

For the primary tree, let $r = 2$ and let $k = \frac{1}{2}B$. There is no reason to change the leaf parameter k from the value suggested in [6], since k controls the spread of objects at the bottom level of the tree. The branching parameter should be set to $r = 2$ to make the primary tree a binary tree. Since the whole primary tree will reside in memory, increased fanout is unnecessary.

Storing the primary tree is trickier for the MTII, since one block will hold more than one nodes. For each node, a node id, left slab boundary, and right slab boundary must be stored, which requires only three units of data. Blocks store tree nodes in a top-down, breadth-first fashion: start at the root, and store each successive level of the tree left, ordering nodes least to greatest for their intervals. The ordering and slab boundaries will inherently denote parent-child relationships. For example, a root node may have the interval [0, 1,000]. Its two children might have [0, 400] and [400, 1,000]. When reading the nodes the jump from a right endpoint of 1,000 to a left endpoint of 0 denotes a new level in the tree. Since the root contains all nodes between 0 and 1,000, the 2 nodes read after the root must be its children. Figure 3.6 illustrates the drastic reduction in storage size between the TII and the MTII.

Uncertain objects are stored in the same way as for the TII, using secondary structures: the left and right endpoint lists are stored externally in blocks. X-bounds for each primary tree node are stored in a copy tree. For each x-bound x value, another primary tree structure is created as mentioned above, only instead of storing slab boundaries, it stores x-bound pairs as if they were slab boundaries. This x-bound tree is stored the same way as the primary tree. Multiple x-bound trees can be created, one for each x value. This way, a query can load the appropriate x-bound tree and not waste disk I/Os on unnecessary x-bound values. A query will only read blocks for the primary tree and one x-bound tree.

Range queries using the MTII are executed similarly as for the TII. First, the primary tree and appropriate x-bound tree must be preloaded. Note that these structures may remain in memory if multiple queries will be executed. Then the stab ’n grab search is performed, just like for the TII.

Update methods are simpler for the MTII. Inserting an object is the same: find the appropriate node based on intervals and store the object in the secondary structures. The primary tree is not updated because extra endpoints are not stored in the nodes. Once the secondary structures are too large, meaning they use more than one or two blocks of data, then the node can be split using standard binary tree procedures. Intervals and x-bounds must also be updated. Deletion follows similar guidelines as for the TII.

Experimental Results

This section presents an evaluation of our experimental results. We use the probability threshold index as a benchmark against which to test both the standard and memory-loaded threshold interval indexes.

Test Model

The purpose for testing these three indexes is to compare their range query performance. Performance is measured by three primary metrics:

- Number of disk I/Os
- Number of probability calculations
- Runtime (in milliseconds)

Six different performance tests are run. Each test builds the indexes from a common data set and runs range queries on each index. Descriptions are given in Table 3.1. Datasets are generated synthetically. Uncertain objects contain two attributes: an id and a PDF. The PDF interval is determined randomly based on test parameters, given in Table 3.2. X-bounds are calculated for the probability values $\{0.1, 0.3, 0.5, 0.7, 0.9\}$ on each index. The block size is 4,096 bytes.

Each test is run with two types of PDFs. Just like for previous tests against the PTI, one PDF used is a uniform PDF [10]. The second PDF is a multimodal Gaussian distribution, which is significantly more complicated. Each PDF can be stored by left and right endpoints and can be stretched to the appropriate interval length. All probability calculations are performed by using Riemann sums. Thousand rectangles are used for each Riemann sum to keep the average error margin around 0.1 %.

Range queries must also be generated. For each test, 100 queries are generated. Each query interval is random within a given domain. Each query is run against each index, and results for all queries are tabulated aggregately. Except for the `Threshold` test, which varies τ, the probability threshold used is $\tau = 0.3$.

Table 3.1 Performance tests

Test	Description
`Same`	Object uncertainty intervals have the same length
`Different`	Object uncertainty intervals have different length
`Dense`	Many objects overlap
`Sparse`	Objects are spaced out
`Threshold`	Same as `Different`, but varies probability threshold
`Ratio`	Varies query interval length versus object interval length

Table 3.2 Test Parameters

Parameter	Same	Different	Dense	Sparse	Ratio
Num objects	10,000	10,000	10,000	10,000	10,000
Min object value	0	0	0	0	0
Max object value	10,000	10,000	1,000	1,000,000	10,000
Min PDF length	100	50	1	1	10
Max PDF length	100	500	100	10	10

Unfortunately, no optimal minimum bounding rectangle (MBR) strategy is proposed for the PTI [10]. Our PTI sorts all objects by their left endpoints and partitions them into leaves for bulk loading. This would most likely represent a "perfect" PTI. However, a more practical PTI goes through insertions and deletions, which will stretch MBRs. For our tests, we separate ten objects from the dataset and insert them into random leaves during bulk loading. This more "practical" PTI is used for comparison testing against the TII and the MTII. Remember, since the TII and MTII are interval trees, they do not experience the same skew problems as the PTI.

Effect of Object Spread

The `Same`, `Different`, `Dense`, and `Sparse` tests all test the spread of uncertain objects. Figure 3.7 gives results for these tests. It is clear that object spread and size significantly affects performance. All indexes have worse performance for objects of different lengths and for densely clustered objects. What is interesting is the difference in performance metrics. The PTI uses fewer disk I/Os than both the TII and the MTII. Although for objects of the same size the MTII and the PTI are comparable for uniform PDFs, the TII and MTII generally use about 1.5–2 times as many disk I/Os. The PTI is far surpassed, however, in regards to the number of probability calculations. Threshold indexes typically use only half to a third of the number of calculations as the PTI. This number is most staggering for sparse indexes: the TII and MTII make relatively no calculations. Overall, the total runtime favors threshold indexes for complicated probability functions, particularly the MTII.

The trends between uniform and multimodal PDFs are generally the same for disk I/Os and probability calculations. This is not too surprising, because PDF shape

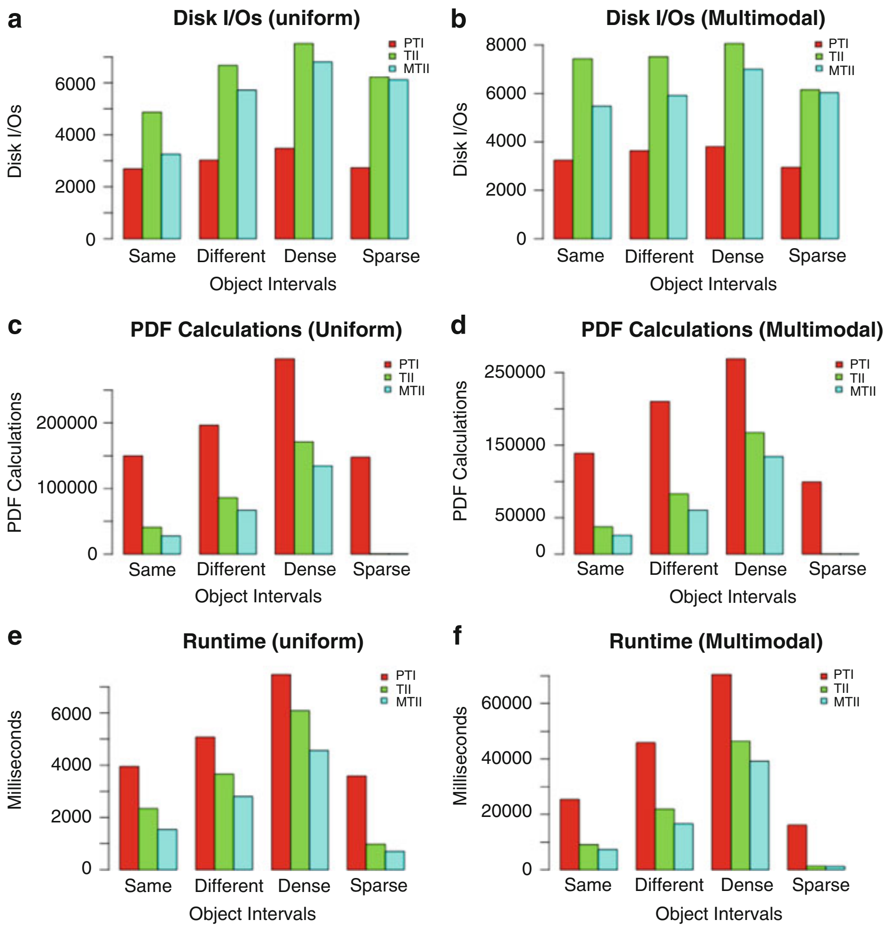

Fig. 3.7 Performance results for `Same`, `Different`, `Dense`, and `Sparse` tests. (**a**) Disk I/Os for uniform PDFs. (**b**) Disk I/Os for multimodal PDFs. (**c**) Uniform PDF calculations. (**d**) Multimodal PDF calculations. (**e**) Runtime for uniform PDFs. (**f**) Runtime for multimodal PDFs

has only a small affect on index structure. The major difference is in total runtime, as seen in Fig. 3.7e, f. Since the multimodal PDF is more complicated, calculations take longer. Thus, the margin by which the TII and MTII outperform the PTI is much larger for multimodal PDFs than for uniform PDFs.

Effect of Probability Threshold

The `Threshold` test uses the same test parameters as the `Different` test, but it runs the query set on multiple probability thresholds. Figure 3.8 gives the

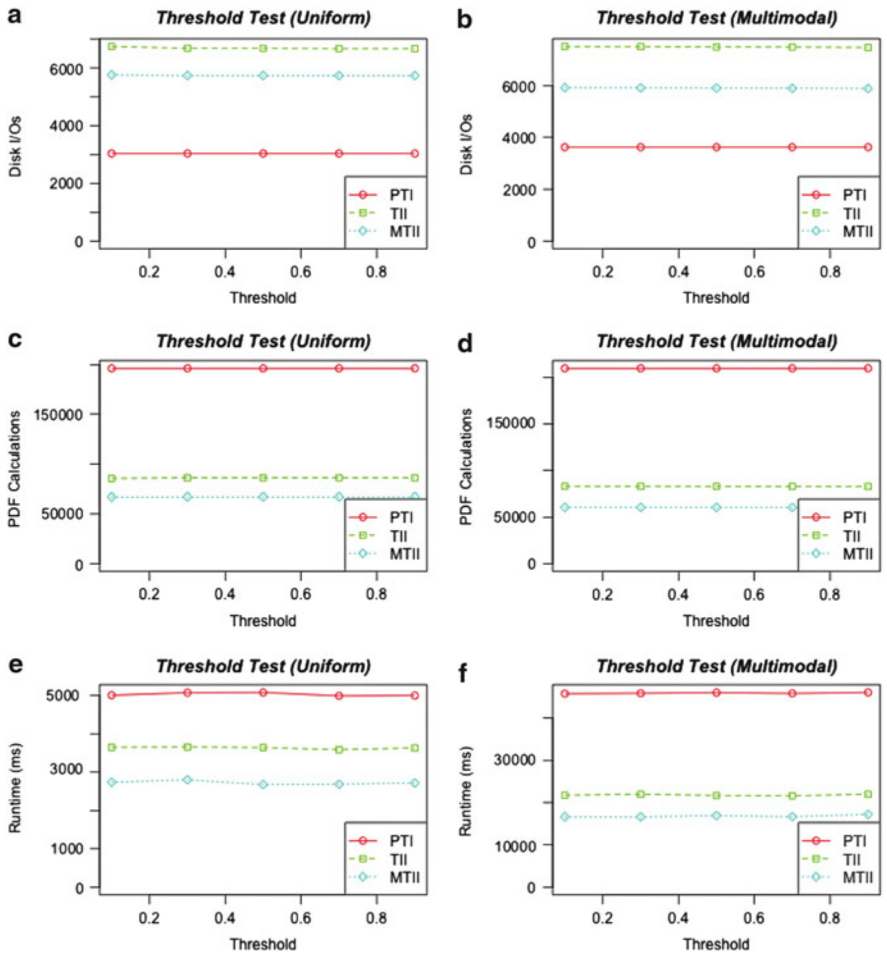

Fig. 3.8 Performance results for the `Threshold` test. (**a**) Disk I/Os for uniform PDFs. (**b**) Disk I/Os for multimodal PDFs. (**c**) Uniform PDF calculations. (**d**) Multimodal PDF calculations. (**e**) Runtime for uniform PDFs. (**f**) Runtime for multimodal PDFs

results. Consistently, probability threshold does not have much effect on any of the indicators of performance, for either PDF.

Effect of Query Interval Size

The `Ratio` test investigates the effect of query size relative to the objects' interval sizes. It runs random queries with a fixed query interval length for different ratios of query interval length to object interval length. For example, a ratio value of 3

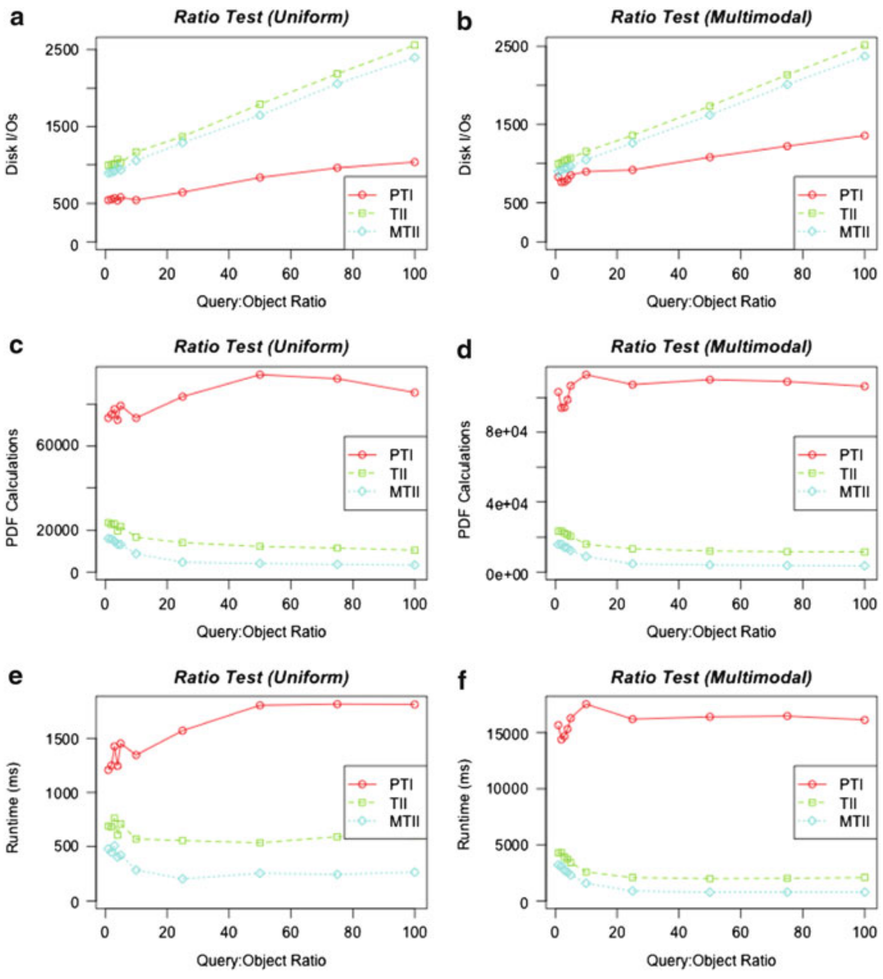

Fig. 3.9 Performance results for the `Ratio` test. (**a**) Disk I/Os for uniform PDFs. (**b**) Disk I/Os for multimodal PDFs. (**c**) Uniform PDF calculations. (**d**) Multimodal PDF calculations. (**e**) Runtime for uniform PDFs. (**f**) Runtime for multimodal PDFs

means that the query interval length is 3 times as large as the object interval length. Each object has a constant interval length of 10 to make ratios consistent. Indexes are built only once, and 100 queries are constructed for each ratio value.

Figure 3.9 gives the results for the `Ratio` test. Results are consistent with the other tests' results. It is interesting to note what happens as the query interval ratio is increased. Naturally, the number of disk I/Os for each index increases linearly, since more objects are fetched for a larger query interval. However, the trend for probability calculations diverge between the PTI and the TII and MTII. For the PTI, the number of calculations appears somewhat parabolic: it generally increases until the ratio value is approximately 50 for uniform PDFs and 10 for multimodal PDFs, after which the number decreases. For the TII and MTII, the number of probability calculations decreases asymptotically towards a very low constant value. This value is about 10,000 for the TII and about 3,000 for the MTII for both PDFs. The runtimes level off after a ratio value of 10 for threshold indexes and about 50 for the PTI with uniform PDFs and about 30 for the PTI with multimodal PDFs. For each index and PDF, runtime always levels off.

These results imply that threshold indexing is superior when query sizes are large in respect to the average object size. This benefit is greatest when query sizes are about ten times as large as object sizes. It also shows again that the MTII is more efficient than the TII.

Conclusion

In this paper, we present threshold interval indexing, a new strategy for indexing complicated uncertain continuous data of one dimension. We present two structures: a standard threshold interval index and a more efficient memory-loaded variant. The key advantage of threshold interval indexing over existing strategies, such as the probability threshold index, is that it handles complicated PDFs much more efficiently because it handles balance better with its intervals.

There are many more opportunities for further research on uncertain data. Specifically, future research should focus more on exploring the effects of different types of PDFs on uncertain indexes. PDFs should not be overlooked, because they are inseparable from what makes the data uncertain. Perhaps certain rectangle management strategies could improve the performance of the PTI, such as the priority R-tree [8] or the segment R-tree [18]. Although our paper focuses on range queries, these indexes could also be used for joins [11, 14]. It is also important for uncertain data strategies to be incorporated into database management systems [5, 28]. Parallelization should also be explored further.

References

1. Agarwal PK, Cheng SW, Tao Y, Yi K (2009) Indexing uncertain data. In: Symposium on principles of database systems (PODS), Providence, pp 137–146
2. Aggarwal CC (2009a) An introduction to uncertain data algorithms and applications. In: Managing and mining uncertain data. Springer, New York, pp 2–8
3. Aggarwal CC (2009b) On clustering algorithms for uncertain data. In: Managing and mining uncertain data. Springer, New York, pp 389–406
4. Aggarwal CC, Yu PS (2008) On indexing high dimensional data with uncertainty. In: IEEE international conference on data engineering (ICDE), Cancun
5. Agrawal P, Benjelloun O, Sarma AD, Hayworth C, Nabar S, Sugihara T, Widom J (2006) Trio: a system for data, uncertainty, and lineage. In: Proceedings of the international conference on very large data bases (VLDB), Seoul, pp 1151–1154. Demonstration Description

6. Arge L, Vitter JS (1996) Optimal dynamic interval management in external memory. In: Proceedings of the IEEE symposium on foundations of computer science (FOCS), Burlington, pp 560–569. Extended abstract
7. Arge L, Vitter JS (2003) Optimal external memory interval management. SIAM J Comput 32(6):1488–1508
8. Arge L, de Berg M, Haverkort HJ, Yi K (2004) The priority r-tree: a practically efficient and worst-case optimal r-tree. In: SIGMOD conference, Paris, pp 347–358
9. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The r*-tree: an efficient and robust access method for points and rectangles. In: SIGMOD conference, Atlantic City, pp 322–331
10. Cheng R, Xia Y, Prabhakar S, Shah R, Vitter JS (2004) Efficient indexing methods for probabilistic threshold queries over uncertain data. In: Proceedings of the international conference on very large data bases (VLDB), Toronto, pp 876–887
11. Cheng R, Xia Y, Prabhakar S, Shah R, Vitter JS (2006) Efficient join processing over uncertain data. In: Proceedings of the 15th ACM international conference on information and knowledge management, Arlington, pp 738–747
12. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms, 2nd edn. Massachusetts Institute of Technology, Cambridge
13. Dellis E, Seeger B (2007) Efficient computation of reverse skyline queries. In: Proceedings of the international conference on very large data bases (VLDB), Vienna, pp 291–302
14. Enderle J, Hampel M, Seidl T (2004) Joining interval data in relational databases. In: SIGMOD conference, Paris, pp 683–694
15. Hua M, Pei J, Zhang W, Lin X (2008) Ranking queries on uncertain data: a probabilistic threshold approach. In: SIGMOD conference, Vancouver, pp 673–686
16. Jensen CS, Lin D, Ooi BC (2004) Query and update efficient b+−tree based indexing of moving objects. In: Proceedings of the international conference on very large data bases (VLDB), Toronto, pp 768–779
17. Kanellakis PC, Ramaswamy S, Vengroff DE, Vitter JS (1993) Indexing for data models with constraints and classes. In: Symposium on principles of database systems (PODS), Washington, DC
18. Kolovson CP, Stonebraker M (1991) Segment indexes: dynamic indexing techniques for multi-dimensional interval data. In: SIGMOD conference, Denver, pp 138–147
19. Kriegel HP, Kunath P, Pfeifle M, Renz M (2006) Probabilistic similarity join on uncertain data. In: Proceedings of the 11th international conference on database systems for advanced applications (DASFAA), Singapore, pp 295–309
20. Kriegel HP, Bernecker T, Renz M, Zuefle A (2009) Probabilistic join queries in uncertain databases. Managing and mining uncertain data. Springer, New York, pp 257–298
21. Lan Y, Papoian GA (2007) Evolution of complex probability distributions in enzyme cascades. J Theor Biol 248:537–545
22. Lian X, Chen L (2008) Monochromatic and bichromatic reverse skyline search over uncertain databases. In: SIGMOD conference, Vancouver
23. Manolopoulos Y, Theodoridis Y, Tsotras VJ (2000) Access methods for intervals, chap 4. In: Advanced database indexing. Kluwer, Boston
24. Pei J, Jiang B, Lin X, Yuan Y (2007) Probabilistic skylines on uncertain data. In: Proceedings of the international conference on very large data bases (VLDB), Vienna
25. Pei J, Hua M, Tao Y, Lin X (2008) Query answering techniques on uncertain and probabilistic data. In: SIGMOD conference, Vancouver, pp 1357–1364. Tutorial summary
26. Prabhakar S, Shah R, Singh S (2009) Indexing uncertain data. In: Managing and mining uncertain data. Springer, New York, pp 299–325
27. Singh S, Mayfield C, Prabhakar S, Shah R, Hambrusch S (2007) Indexing uncertain categorical data. In: IEEE international conference on data engineering (ICDE), Istanbul, pp 616–625
28. Singh S, Mayfield C, Mittal S, Prabhakar S, Hambrusch S, Shah R (2008) Orion 2.0: native support for uncertain data. In: SIGMOD conference, Vancouver, pp 1239–1242. Demonstration Description

29. Soliman MA, Chang KC-C, Ihab F. Ilyas (2007) Top-*k* query processing in uncertain databases. In: IEEE international conference on data engineering (ICDE), Istanbul
30. Tao Y, Cheng R, Xiao X, Ngai WK, Kao B, Prabhakar S (2005) Indexing multi-dimensional uncertain data with arbitrary probability density functions. In: Proceedings of the international conference on very large data bases (VLDB), Trondheim
31. Wolfson O, Sistla AP, Chamberlain S, Yesha Y (1999) Updating and querying databases that track mobile units. Distrib Parallel Databases (Special issue on Mob Data Manage Appl) 7(3):257–387
32. Yi K, Li F, Kollios G, Srivastava D (2008) Efficient processing of top-*k* queries in uncertain databases. In: IEEE international conference on data engineering (ICDE), Cancun
33. Yu J, Yang MS, Hao P (2009) A novel multimodal probability model for cluster analysis. In: RSKT '09: proceedings of the 4th international conference on rough sets and knowledge technology. Springer, Berlin/Heidelberg, pp 397–404. doi:http://dx.doi.org/10.1007/978-3-642-02962-2_50
34. Zhang M, Chen S, Jensen CS, Ooi BC, Zhang Z (2009) Effectively indexing uncertain moving objects for predictive queries. In: Proceedings of the international conference on very large data bases (VLDB), Lyon